ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection

Xiang Ni, Esteban Meneses, Nikhil Jain, Laxmikant V. Kalé
Department of Computer Science, University of Illinois at Urbana-Champaign
{xiangni2, emenese2, nikhil, kale}@illinois.edu

ABSTRACT

As machines increase in scale, many researchers have predicted that failure rates will correspondingly increase. Soft errors do not inhibit execution, but may silently generate incorrect results. Recent trends have shown that soft error rates are increasing, and hence they must be detected and handled to maintain correctness. We present a holistic methodology for automatically detecting and recovering from soft or hard faults with minimal application intervention. This is demonstrated by ACR: an automatic checkpoint/restart framework that performs application replication and automatically adapts the checkpoint period using online information about the current failure rate. ACR performs an application- and user-oblivious recovery. We empirically test ACR by injecting failures that follow different distributions for five applications and show low overhead when scaled to 131,072 cores. We also analyze the interaction between soft and hard errors and propose three recovery schemes that explore the trade-off between performance and reliability requirements.

Keywords

Fault-tolerance, silent data corruption, checkpoint/restart, redundancy

1. INTRODUCTION

In recent times, the HPC community has seen a stellar growth in the capability of high-end systems. Machines such as IBM Blue Gene/Q and Cray XK7 have a peak performance that reaches to the tens of petaflops. Since the frequency of CPUs has been limited in recent years, these systems have increased processing power by increasing the number of cores. However, the increase in the number of system components required to build these machines has had a negative impact on the reliability of the system as a whole. If these trends persist, large systems in the near future may experience hard failures very frequently [5, 18].

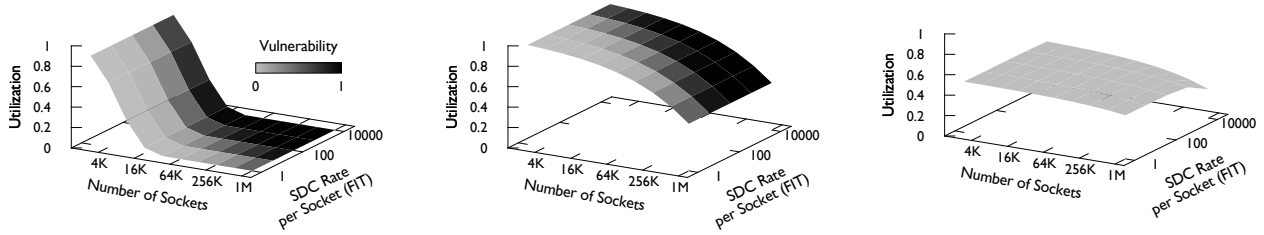
Soft errors are becoming more prevalent as feature sizes decrease along with supply voltages to chips [25]. As systems become more energy-efficient, chips may approach near-threshold operation to reduce the power required. Previous studies have shown a strong correlation between the increase in soft error rate and the decrease in device sizes and operating voltages [9]. This same study suggests that the soft error rate may reach very high levels, to the point that an undetected soft error may occur once per day in a single chip. The most insidious form of soft error is silent data corruption (SDC) [25]. For mission-critical applications, having a guarantee that the data was not silently corrupted may be very important. Even today's systems face a modest amount of soft errors. For example, ASC Q at LANL experienced on average 26.1 radioactivity-induced CPU failures per week [23].

The common approach currently is to tolerate intermittent faults by periodically checkpointing the state of the application to disk and restarting when needed. However, as hard failure rates increase along with machine sizes, this approach may not be feasible due to high overheads. If the data size is large, the expense of checkpointing to disk may be prohibitive, and may incur severe forward-path overheads. At the possible cost of memory overhead, recent libraries for checkpointing have successfully explored alternative local storage resources to store checkpoint data [2, 24, 30].

Although checkpoint/restart strategies may be effective for hard faults, SDC can not be detected using them. Detection of SDC is a difficult problem and most traditional fault tolerance approaches applied to HPC fail at addressing this problem. One possible solution, which has been shown to be effective in the past, is to use redundant computation to detect SDC and correct them. This approach is beneficial because it is universally applicable and is well-established as a solid approach. However, due to its perceived high cost, it is only recently been explored for HPC. Analytical studies have shown that if the failure rate is sufficiently high, introducing redundancy to handle errors may actually increase the overall efficiency in the HPC realm [8, 10]. Other work has shown that redundancy is an effective methodology for detecting SDC in HPC applications [11].

In order to better grasp the problem, we model the system utilization and system vulnerability (the probability of finishing execution with an incorrect result) as the number of sockets increase and SDC rate increases. Figure 1a shows the results of varying these parameters without any fault tolerance. Note that, as the socket count increases from 4K to 16K, the utilization rapidly declines to almost 0. With hard-error resilience (shown in Figure 1b) using checkpoint/restart, the utilization increases substantially, but still

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13 November 17-21, 2013, Denver, Colorado, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503266



(a) No fault-tolerance protection (b) Hard-error checkpoint-based protection (c) ACR with protection from SDC and hard error

Figure 1: Overall system utilization and vulnerability to SDC with different fault tolerance alternatives (for a job running 120 hours). ACR offers holistic protection using scalable mechanisms against SDC and hard errors.

drops after 64K sockets. However, since checkpoint/restart cannot detect SDC, the vulnerability remains very high. To mitigate both these problems, we present ACR: a scalable, automatic checkpoint/restart framework that can detect and correct both SDC and hard errors. As shown in Figure 1c, by using our framework the system vulnerability disappears and the utilization remains almost constant; the utilization penalty, which seems significant at small scale, is comparable to other cases at scale.

We believe, to reach the exascale computing realm effectively, we must develop holistic solutions that cover all the types of failures that may occur in the system. Although hard failures may be detectable by external system components, soft errors remain elusive. Hence, software solutions that address both problems in an integrated fashion are needed. ACR does exactly this, and also utilizes a novel mechanism to interact with applications for checkpointing based on observed failure rates.

By developing this framework and empirically evaluating it under various failure scenarios, we make the following contributions:

- We present a novel design for an automatic checkpoint/restart mechanism that tolerates both SDC and hard errors, and can adaptively adjust the checkpointing period (§2, §3, §4).
- We present a distributed algorithm for determining checkpoint consensus asynchronously, and show empirically that it causes minimal application interference (§2).
- We present three distinct recovery schemes in ACR that explore the tradeoff between performance and reliability. Using a model we have developed, we analytically study for these schemes the interaction between hard-error recovery and soft-error vulnerability at large scales (§2, §5).
- We demonstrate use of topology-aware mapping to optimize communication, and empirically show that this results in significant speedup during checkpointing and restart (§4, §6).
- We evaluate ACR by showing for five mini-applications, written in two programming models, on 131,072 cores that the framework is highly scalable and adapts to dynamic behavior (§6).

2. AUTOMATIC CHECKPOINT RESTART

In this section, we describe the Automatic Checkpoint/Restart (ACR) framework, a low-overhead framework that aims to provide protection from both SDC and hard errors to applications. To handle failures efficiently, ACR automatically checkpoints at an adaptive rate on the basis of failure history. If failures occur, based on the type of error, ACR enacts corrective measures and performs an automatic restart.

2.1 Replication-enhanced Checkpointing

ACR uses checkpointing and replication to detect SDC and enable fast recovery of applications from SDC and hard errors. When a user submits a job using ACR, a few nodes are marked as spare nodes and are not used by the application, but only replaces failed nodes when hard errors occur. The rest of the nodes are equally divided into two partitions that execute the same program and checkpoint at the same time. We refer to these two partitions as replica 1 and replica 2. Each node in replica 1 is paired with exactly one unique node in replica 2; we refer to these pairs as *buddies*. Logically, checkpointing is performed at two levels: local and remote. When a checkpoint is instigated by the runtime, each node generates a local checkpoint by invoking a serialization framework to save its current state. Programmers are required to write simple functions that enable ACR to identify the necessary data to checkpoint. Local checkpoint of a node in one replica serves as remote checkpoint of the buddy node in another replica.

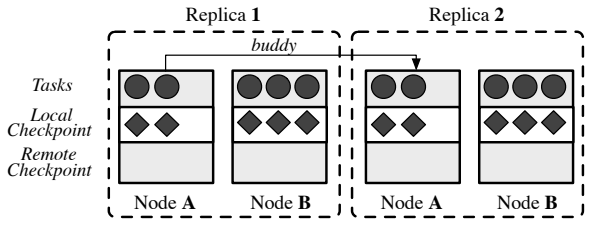


Figure 2: Replication enhanced checkpointing. The buddy of a node is the corresponding node in the other replica.

Hard Error Protection: We assume hard errors follow a fail-stop model, i.e. a failed node does not perform any communication. We call the replica containing the crashed node the *crashed* replica and the other replica the *healthy* replica. After a failure is detected, the buddy node (in the healthy replica) of the crashed node sends its own local checkpoint to the new node (from the spare pool) that replaces the crashed node. Since the paired buddy nodes perform exactly the

same work during forward-path execution, the crashed node can be restarted using the checkpoint of its buddy node on the new node. Every other node in the crashed replica rolls back using the checkpoint stored locally.

Detection and Correction of Silent Data Corruption: In order to detect SDC, every node in the replica 1 sends a copy of the local checkpoint to its buddy node in replica 2. Upon receiving the remote checkpoints from their buddy nodes in replica 1, every node in replica 2 compares the remote checkpoint with its local checkpoint using the same serialization framework used to pack the checkpoint. If a mismatch is found between the two checkpoints, ACR rolls back both the replicas to the previous safely stored local checkpoint, and then resumes the application. Note that the remote checkpoint is sent to the replica 2 only for SDC detection purposes, and hence ACR does not store the remote checkpoint. Figure 2 shows the stable state of nodes during application execution when using ACR.

2.2 Automatic Checkpoint Decision

An important feature of replication-enhanced checkpointing is its ability to reduce recovery overhead in the face of hard errors, which is enabled by automatic checkpointing. When a hard failure occurs, if an immediate checkpoint can be performed in the healthy replica to help the crashed replica recover instead of using the previous checkpoint, the crashed replica can quickly catch up with the progress of the healthy one. Moreover, as online failure prediction [19] becomes more accurate, checkpointing right before a potential failure occurs can help increase the mean time between failures visible to applications. ACR is capable of scheduling dynamic checkpoints in both the scenarios described.

Upon reception of a checkpointing request, ACR can not simply notify every task to store its state. This may lead to a situation where an inconsistent checkpoint is stored, causing the program to hang. For example, in an iterative application, assume task a receives the checkpoint decision at iteration i, and after restart it will wait to receive a message from task b to enter iteration $i+1$. However when task b receives the checkpoint decision, it is already at $i+1$ and has already sent out message c. This in-flight message c will not be stored in the checkpoint anywhere. Thus after restart, task b will be unaware that it needs to resend the message to a and task a will hang at iteration i. This scenario is possible when there is no global synchronization at each iteration and each task progresses at different rates during application execution. ACR ensures the consistency of checkpointing with minimal interference to applications using the following scheme.

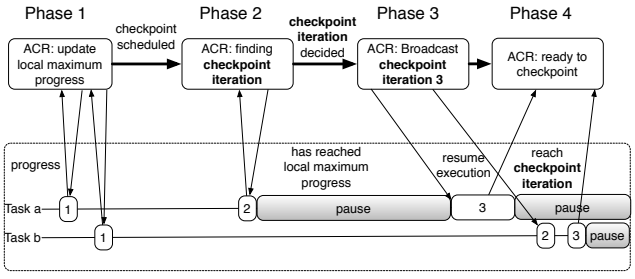


Figure 3: Initialization of automatic checkpointing.

Periodically, each task reports its progress to ACR through a function call. In an iterative application, for example, this function call can be made at end of each iteration. In most cases, when there is no checkpointing scheduled, this call returns immediately. ACR records the maximum progress among all the tasks residing on the same node as shown in Figure 3 (Phase 1). If checkpointing is required, either due to a failure in one of the replicas or based on an observation of the failure history, ACR proceeds to find a safe checkpoint iteration.

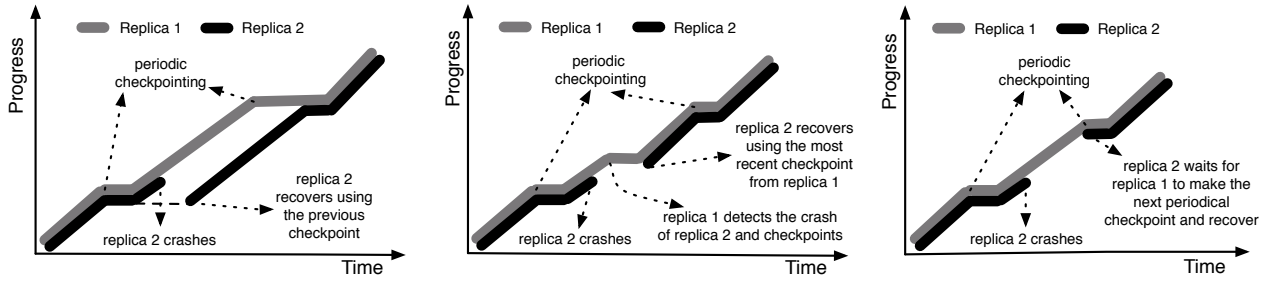
Using the local progress information, ACR begins an asynchronous reduction to find the maximum progress among all the tasks. In the mean time, tasks whose progress has reached the local maximum are temporarily paused to prevent tasks from going beyond the possible checkpoint iteration (Figure 3, Phase 2). Once ACR finds the maximum progress in the system, the checkpoint iteration is decided accordingly. Each task compares its progress with the checkpoint iteration; if its progress has reached the checkpoint iteration, the task is considered ready for the checkpoint and transitions to the pause state if it was in the execution state or remains in the pause state if it was already in the pause state. Otherwise, the computation task will continue or resume execution until it reaches the checkpoint iteration (Figure 3, Phase 3). Eventually, when all the tasks get ready for the checkpoint, checkpointing is initiated (Figure 3, Phase 4). The more frequently the progress function is invoked, the sooner ACR can schedule a dynamic checkpoint.

Adapting to Failures: It has been shown that a fixed checkpoint interval is optimal if the failures follow a Poisson process [7]. However, a study of a large number of failure behaviors in HPC systems [28] has shown that a Weibull distribution is a better fit to describe the actual distribution of failures. An important point to note in this study is that the failure rate often decreases as execution progresses. Dynamically scheduling checkpoints has shown benefits in such scenarios in comparison to a fixed checkpoint interval in an analytical study [4, 20]. Hence, it is important to fit the actual observed failures during application execution to a certain distribution and dynamically schedule the checkpoints based on the current trend of the distribution. To support such adaptivity, ACR provides a mode in which each checkpoint interval is decided based on the distribution of the streaming failures. This is enabled by the automatic checkpointing and automatic recovery in ACR.

2.3 Interaction of Hard Error Recovery and Vulnerability to Silent Data Corruption

Detection and correction of SDC using replication enables ACR to recover from hard failures in different ways. These choices offer *novel trade-offs* that have not been encountered in any framework for HPC. Three resilience schemes may be used in ACR depending on the reliability and performance requirements of an application.

1) Strong Resilience: In this scheme, the crashed replica is rolled back to the previous checkpoint. The restarting process (on the spare node) is the only process in the crashed replica that receives the checkpoint from the other replica, and hence minimal network traffic is generated. Every other node in the crashed replica rolls back using its own local checkpoint. Figure 4a shows the progress chart in which replica 2 is recovered using strong resilience. When a hard error is encountered in replica 2, the crashed replica restarts using the previous checkpoint. Having reaching the next



(a) Strong resilience recovery scheme (b) Medium resilience recovery scheme (c) Weak resilience recovery scheme

Figure 4: Recovery in different resilience levels of ACR. Strong resilience rolls back immediately after a hard error. Medium resilience forces an additional checkpoint and restarts from there. Weak resilience waits until the next checkpoint to restore.

checkpoint period, replica 1 waits for replica 2 to resume application execution.

The advantage of using this scheme is 100% protection from SDC. The execution of applications in the two replicas is always cross-checked. Additionally, restarting the crashed replica is very fast because only one message is sent from the healthy replica to the restarting process. However, the amount of rework being done is large, and it may slow down the application progress.

2) Medium Resilience: This scheme attempts to reduce the amount of rework by forcing the healthy replica to immediately schedule a new checkpoint when a hard error is detected in the crashed replica as shown in Figure 4b. The latest checkpoint is sent from every node of the healthy replica to their buddy nodes in the crashed replica, which may incur relatively higher overhead in comparison to the strong resilience scheme. Moreover, any SDC that occurred between the previous checkpoint and the latest checkpoint will remain undetected. On the positive side, this scheme avoids rework on the crashed replica and hence the two replicas reach the next checkpoint period at similar times in most cases. However, if a hard failure occurs in the healthy replica before the recovery of crashed replica is complete, application needs to rollback either to the previous checkpoint or the beginning of execution. Since the healthy replica schedules the next checkpoint in a very short period, the probability of such a rollback is very low.

3) Weak Resilience: In this scheme, the healthy replica does not take any immediate action to restart the crashed replica when a hard error is detected. Instead, it continues execution until the next checkpoint and thereafter sends the checkpoint to the crashed replica for recovery. This scheme leads to a "zero-overhead" hard error recovery since in most cases the healthy replica does not incur any extra checkpoint overhead to help the crashed replica recover, and the crashed replica does not spend any time in rework. The only exception is when hard failure occurs in the healthy replica before it reaches the next checkpoint time. If the failure happens on the buddy node of the crashed node (though the probability is very low [22, 10]) application needs to restart from the beginning of the execution. Otherwise application needs to restart from the previous checkpoint. Typically, the system is left unprotected from SDC for the entire checkpoint interval. Figure 4c shows the event chart for this resilience scheme. Assuming a large rework time, Figure 4 suggests that this scheme should be the fastest to finish application execution.

2.4 Control Flow

Figure 5 presents a study of application execution using ACR with different reliability requirements. In each scenario, execution begins with the division of the allocated nodes into two replicas each of which performs the same application task.

Figure 5(a) presents the scenario in which an application only requires support for handling hard errors. No periodic checkpointing is needed in this scenario. When a node crash is detected at *T2*, replica 2 schedules an immediate checkpoint, and sends the checkpoint to replica 1. This allows replica 1 to continue on the forward path without rollback.

Figures 5(b,c,d) present the general scenario in which both silent data corruptions and hard errors may occur in the system. In all these scenarios, periodic local checkpointing is performed (e.g. at time *T1*, *T3* etc.). When a hard failure occurs at time *T2*, in the strong resilience scheme shown in Figure 5(b), replica 2 sends its SDC-free local checkpoint at *T1* to the restarting process in replica 1 to help it recover. The application is fully protected from SDC in this scenario. However, in Figure 5(c) with medium resilience, an immediate checkpoint is performed in replica 2 when a failure occurs at time *T2*. Replica 1 is recovered using this new checkpoint. As such, at time *T3*, only SDC that occurred after *T2* will be detected. In weak resilience scheme of Figure 5(d), replica 2 continues execution until the next scheduled checkpoint time *T3*, and then sends this checkpoint to replica 1 for recovery. Although this scheme incurs zero-overhead in the case of a hard failure, the application is not protected from SDC from time *T1* to *T3*.

3. DESIGN CHOICES

During the design process of ACR, we evaluated alternative methods for different components of the framework, and selected the ones most suited to our needs. In this section, we present those design choices and their trade-offs relative to the alternatives.

1) Ensuring consistent states. To enable the recovery of a crashed replica from a hard error using information from the healthy replica, it is necessary that the processes in the two replicas are interchangeable, i.e. for every process in replica 1 there is a process in replica 2 that has the same application state. ACR makes use of coordinated checkpointing to ensure this consistency, and hence does not require

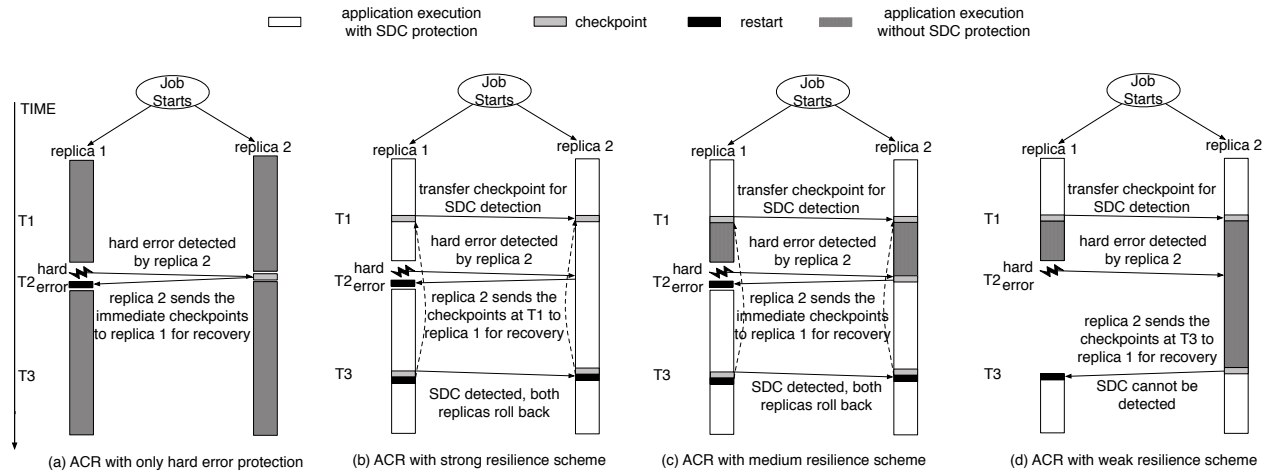

application execution with SDC protection
checkpoint
restart
application execution without SDC protection
Job Starts
TIME
replica 1
replica 2
T1
hard error detected by replica 2
hard error
T2
replica 2 sends the immediate checkpoints to replica 1 for recovery
T3
(a) ACR with only hard error protection
transfer checkpoint for SDC detection
replica 2 sends the checkpoints at T1 to replica 1 for recovery
SDC detected, both replicas roll back
(b) ACR with strong resilience scheme
(c) ACR with medium resilience scheme
replica 2 sends the checkpoints at T3 to replica 1 for recovery
SDC cannot be detected
(d) ACR with weak resilience scheme

Figure 5: The control flow of ACR with different reliability requirements.

any communication or synchronization between replicas unless a hard error occurs.

An alternative approach to maintain the consistent state between replicas is by cloning messages. Libraries such as rMPI [10] and P2P-MPI [13], which provide replication-based reliability to MPI applications, provide reliability support by ensuring that if an MPI rank dies, its corresponding MPI rank in the other replica performs the communication operations in its place. This approach requires the progress of every rank in one replica to be completely synchronized with the corresponding rank in the other replica before and after the hard error. Such a fine-grained synchronization approach may hurt application performance, especially if a dynamic application performs a large number of receives from unknown sources. In fact, in such scenarios the progress of corresponding ranks in the two replicas must be serialized to maintain consistency. For a message-driven execution model in which the execution order is mostly non-deterministic, this approach is certainly not optimal.

2) Who detects silent data corruption? In ACR, the runtime is responsible for transparently creating checkpoints and their comparison to detect SDC. Algorithmic fault tolerance is an alternative method based on redesigning algorithms using domain knowledge to detect and correct SDC [3]. Use of containment domains [6] is a programming-construct methodology that enables applications to express resilience needs, and to interact with the system to tune error detection, state preservation, and state restoration. While both these approaches have been shown to be scalable, they are specific to their applications. One may need to have in-depth knowledge of the application domain and make significant modifications to the code in order to use them. In contrast, a runtime-based method is universal and works transparently with minimal changes to the application. Hence, we use this strategy in ACR.

3) Methods to detect silent data corruption. Similar to *ensuring consistent states*, an alternative method to detect SDC is to compare messages from the replicas [11]. If a message is immediately sent out using the corrupted memory region, early detection of SDC is possible using this scheme. However, a major shortcoming of message-based error detection is the uncertainty of error detection – if the data effected by SDC remains local, it will not be detected. Moreover, even when corruption has been detected, it may be difficult to correct the corrupted data on the source process if the corruption was not transient or was used in the computation. Checkpoint-based SDC detection does not suffer from any of these issues; given the synergy with the hard-error recovery method, it is the appropriate choice for ACR.

4) Redundancy model. Based on dual redundancy, ACR requires re-executing the work from the last checkpoint if one SDC is detected. Alternatively, triple modular redundancy (TMR) is a popular method to provide resilience for applications that have real-time constrains. In TMR, the results processed by the three redundant modules are passed by a voting system to produce a single output and maintain consistency. The trade off to consider between dual redundancy and TMR is between re-executing the work or spending another 33% of system resources on redundancy. We have chosen the former option assuming good scalability for most applications and relatively small number of SDCs. Dual redundancy, as a fault tolerance alternative, requires to invest at least 50% of the system's utilization. This is a considerable price to pay upfront to recover from SDCs, but it is a general-purpose solution. Additionally, it has been shown that replication outperforms traditional checkpoint/restart for scenarios with high failure rates [10].

5) Checkpointing level. Checkpointing can be performed either at the kernel or user level. Kernel-level checkpointing like BLCR [14] dumps all the system state and application data during checkpointing. As a result it can quickly react to a failure prediction. In contrast, user-level checkpointing such as SCR [24] is triggered by the application at a certain interval. Compared to kernel-level checkpointing, it can reduce the checkpoint size since the process state and buffered messages are not stored. ACR performs user-level checkpointing but with the simplicity and flexibility advantages of the kernel-level scheme. The checkpointing in ACR is equivalent to the combination of LOCAL and PARTNER levels in SCR. Each checkpoint is invoked by the runtime at a safe point specified by the user in order to store the minimal state needed. But we allow the interval between checkpoints to be dynamically adjusted to the observed failure rate without user involvement.

4. ADAPTATION AND OPTIMIZATIONS

An adaptation of ACR to Charm++ has been performed to validate it on real systems executing applications of various types. In order to support ACR, we have made use of some existing features in Charm++ and added new ones. Important optimizations to boost performance and reduce overheads have been performed.

4.1 Implementation Details

Replication. To support replication, we have augmented Charm++ with support for transparent partitioning of allocated resources. On a job launch, ACR first reserves a set of spare nodes (§ 2) to be used in event of failure. The remaining nodes are divided into two sets that constitute the two replicas. The application running in each replica is unaware of the division and executes independently in each replica. In addition to support for regular intra-replica application communication, we have added an API for inter-replica communication that is used by ACR for various purposes.

Checkpointing. Charm++ supports checkpointing to either memory or file system using simple user specified *pup* functions. The *pup* functions use the Pack and UnPack (PUP) framework to serialize/deserialize the state of the application to/from the chosen medium using special PUPer(s).

Handling SDC. We have augmented the PUP framework to support a new PUPer called the *checker*. This PUPer compares the local checkpoint of a node with the remote checkpoint sent to the node by its buddy, and reports if silent data corruption has occurred. *PUPer::checker* also enables a user to customize the comparison function based on their application knowledge. For example, since the floating point math may result in round-off errors, a programmer can set the relative error a program can tolerate. One may also guide the PUP framework to ignore comparing data that may vary between different replicas, but are not critical to the result.

4.2 Optimizations

Simultaneous transmitting checkpoints for comparison or during restart using the weak/medium resilience scheme may saturate the network and result in congestion. We have implemented the following two techniques to reduce network congestion, and hence improve the performance of ACR.

Checksum. A simple but effective solution to network congestion problem is use of a *checksum* to compare the checkpoints. ACR uses the position-dependent Fletcher's checksum algorithm [12] to calculate the checksum of a checkpoint, which is then transmitted to the buddy for comparison. While the use of checksums reduces the load on the network, it increases the computation cost. Instead of a single instruction required to copy the checkpoint data to a buffer if the full checkpoint is sent, 4 extra instructions are needed to calculate the checksum. Assuming a system that has the communication cost per byte of β and computation cost of γ per byte, the difference in cost of the two schemes is $(\beta - 4\gamma) \times n$. Hence, using the checksum shows benefits only when $\gamma < \frac{\beta}{4}$.

Topology-aware mapping. ACR implements topology-aware task mapping to reduce network congestion during restart and checkpoint comparison (if checksum is not used). Consider the default mapping of replicas onto 512 nodes of Blue Gene/P running in shared-memory mode (Figure 6(a)). Only the mapping for the front plane ($Y = 0$) is shown for ease of understanding. Replica 1 is allocated the block of nodes that constitute the left half of the allocations, whereas replica 2 is allocated the right half. During checkpointing, node i of replica 1 sends a message to node i of replica 2. Using the default mapping, the nodes are laid out such that the path taken by checkpoints sent by nodes in each column overlaps with the path traversed by checkpoints sent by nodes in their row in every other column. In Figure 6(a), this overlap is represented by tagging each link with the number of messages that pass through them during checkpointing. Even if the torus links are considered, the overlap on links exist albeit in lower volume. In effect, the links at the bisection of replica 1 and replica 2 become the bottleneck links; the loads on these bottleneck links are determined by the number of columns. On BG/P, the default mapping is $TXYZ$ in which ranks increase slowest along Z dimension; hence the two replicas are divided along the Z dimension and the load on bottleneck links is proportional to the length of Z dimension.

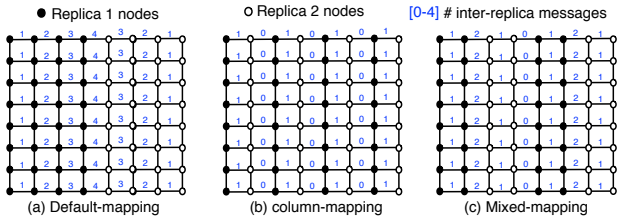


Figure 6: Mapping schemes and their impact on inter-replica communication: the number on the links is the number of checkpoint messages that will traverse through those links.

The excess load on the bottleneck link can be reduced by using an intelligent mapping that places the communicating nodes from the two replicas close to each other. Consider the *column*-mapping of the two replicas in Figure 6(b) that alternatively assigns the columns (and the corresponding Z planes that are not shown) to replica 1 and replica 2. This kind of mapping eliminates the overlap of paths used by inter-replica messages, and is best in terms of network congestion. However, providing disjoint mapping for a replica may interfere with application communication and result in slow progress for communication-intensive applications. Additionally, placing the buddy nodes close to each other increases the chances of simultaneous failures if the failure propagation is spatially correlated. In such scenarios, one may use *mixed*-mapping in which chunks of columns (and the corresponding planes) are alternatively assigned to the replicas as shown in Figure 6(c).

Another way to reduce network congestion is to use asynchronous checkpointing [26] that overlaps the checkpoint transmission with application execution. We leave implementation and analysis of this aspect for future work.

5. MODELING PERFORMANCE AND RELIABILITY

A fundamental question when using checkpoint/restart is how often to checkpoint. Frequent checkpoints will imply

less work to be recovered in case of a failure, but it will incur high overhead because of the more time spent on checkpoints. This section presents a model to help understand the performance and reliability difference for the three resilience schemes in ACR. The presented model represents a system with a number of parameters and defines several equations to compute relevant values: optimum checkpoint period, total execution time and probability of undetected silent data corruptions. Additionally, the model allows us to understand how ACR will scale and perform in different scenarios.

The model extends the theoretical framework presented in the literature [7] by incorporating SDC in the equations, and three different levels of resilience recovery schemes. We assume failures follow the Poisson process. Parameters used in the model are listed in Table 1. These parameters include application-dependent parameters (W, δ,R_H,R_S), system-dependent parameters (M_H, M_S, S), and the output of the model (τ, T, T_S, T_M, T_W).

	Description		Description
W	Total computation time	τ	Optimum checkpoint period
δ	Checkpoint time	S	Total number of sockets
R_H	Hard error restart time	T	Total execution time
R_S	Restart time on SDC	T_S	T strong resilience
M_H	Hard error MTBF	T_M	T medium resilience
M_S	SDC MTBF	T_W	T weak resilience

Table 1: Parameters of the performance model.

Since total execution time is the main variable of interest which we are trying to minimize, we use the following equation to describe the different components:

$$T = T_{Solve} + T_{Checkpoint} + T_{Restart} + T_{Rework}$$

where T_{Solve} is the useful computation time, $T_{Checkpoint}$ is the time spent exclusively on checkpointing, $T_{Restart}$ is the time spent in restarting applications for execution after detecting any type of error, and T_{Rework} stands for the time spent in re-executing the work after both SDC and hard errors. The total checkpointing time is simply the product of the individual checkpoint time and the number of checkpoints:

$$\Delta = T_{Checkpoint} = \left(\frac{W}{\tau} - 1\right)\delta$$

The total restart time is similarily the product of individual restart time and the number of restarts:

$$R = T_{Restart} = \frac{T}{M_H}R_H + \frac{T}{M_S}R_S$$

In order to represent the three different levels of resilience defined in Section 2, we define an equation for each level. The total execution time for strong resilience level (T_S) uses the fact that a hard error will require the system to rollback immediately to a previous checkpoint. The medium resilience level (whose total execution time is T_M) will checkpoint right after the hard error, so on average, half that checkpoint interval the system is unprotected against SDC. Finally, the weak resilience level (T_W represents the total execution time) will leave the whole checkpoint period unprotected against SDC. The equations for these variables are presented below. As discussed in Section 2.3, the application may need to rollback to the previous checkpoint when a hard failure occurs in the healthy replica using the weak resilience scheme. P is the probability for more than one failure in a checkpoint period. Note that this is a loose upper bound on the probability to rollback; we assume that atleast one of the multiple failures happens in the healthy replica.

$$P = 1 - \exp(-\frac{\tau+\delta}{M_H})(1 + \frac{\tau+\delta}{M_H})$$

$$T_S = W + \Delta + R + \frac{T_S}{M_H}\left(\frac{\tau+\delta}{2}\right) + \frac{T_S}{M_S}(\tau+\delta)$$

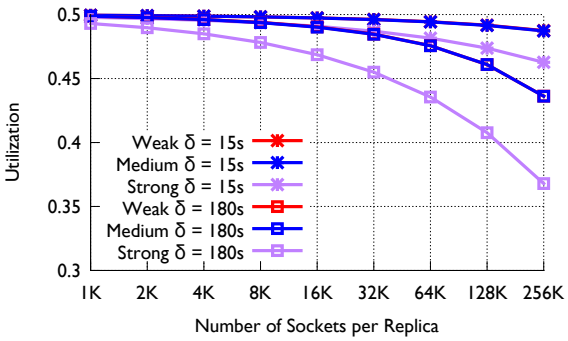
$$T_M = W + \Delta + R + \frac{T_M}{M_H}\delta + \frac{T_M}{M_S}(\tau+\delta)$$

$$T_W = W + \Delta + R + \frac{T_S}{M_H}\left(\frac{\tau+\delta}{2}\right)P + \frac{T_W}{M_S}(\tau+\delta)$$

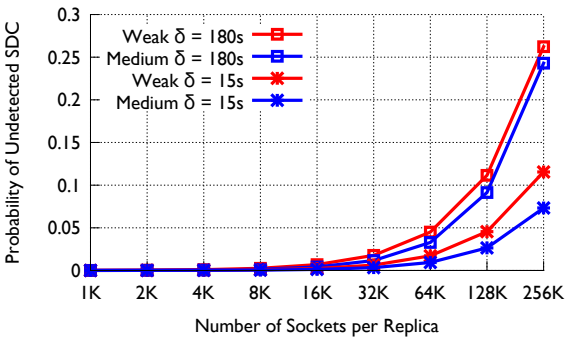
Using these formulae, we calculate the optimal checkpoint interval for the three resilience schemes and use the best total execution time for further analysis.

Performance and Protection: We define the *utilization* of the system as the portion of the time that is devoted to do useful work: $\frac{W}{T}$. The complement of the utilization is the overhead of the fault tolerance approach. This overhead includes the checkpoint time, restart time, rework time and the utilization loss due to replication. Figure 7a shows the utilization of different schemes with different checkpoint time from 1K sockets to 256K sockets per replica. The checkpoint time projected for exascale machine ranges from seconds to minutes [18]. Thus, we choose δ to be 180s and 15s to represent large and small checkpoints respectively (since one of the dominant factors in δ is the checkpoint size [§ 6]). We assume a mean time between hard errors M_H of 50 years (equivalent to the MTBF of Jaguar system [29]) and SDC rate of 100 FIT [1]. For δ of 15s, the efficiency for all the three resilience schemes is above 45% even on 256K sockets. When δ is increased to 180s, the efficiency of the strong resilience scheme decreases to 37% while that of the weak and medium resilience schemes is above 43% using 256K sockets. Note that in weak and medium resilience schemes, the system is left without any SDC protection for some period of time. Hence, based on the application, one may have to sacrifice different amount of utilization to gain 100% protection from SDC.

Figure 7b presents the probability of occurrence of SDC during the period in which the framework does not provide any protection to silent data corruptions using medium and weak resilience schemes for a job run of 24 hours. The results suggest that for low socket count (up to 16K sockets), the probability of an undetected error is very low for the two types of applications we considered. It is also worth noting that even on 64K sockets, the probability of an undetected SDC for the medium resilience scheme is less than 1% (using δ = 15s). These scenarios may be sufficient to meet the resilience requirements of some users with minimal performance loss. However, the probability of an undetected SDC is high on 256K sockets, and users will have to choose the strong resilience scheme with some performance loss to execute a fully protected experiment. For both the cases, the medium resilience scheme decreases the probability of undetected SDC by half with negligible performance loss.



(a) Utilization



(b) Probability of undetected SDC

Figure 7: The utilization and vulnerability of the different recovery schemes for different checkpoint size. Strong resilience scheme detects all the SDCs but results in a loss of 65% utilization. Weak resilience scheme has the best utilization but is more likely to have undetected SDC for a large δ. Medium resilience scheme reduces the likelihood of undetected SDC with little performance loss.

6. EVALUATION

6.1 Setup

We have used various mini-applications including a stencil-based state propagation simulation, a molecular dynamic simulation, a hydrodynamics simulation using an unstructured mesh, and a conjugate gradient solver to evaluate ACR.

Jacobi3D is a simple but commonly-used kernel that performs a 7-point stencil-based computation on a three dimensional structured mesh. We evaluate our framework using a CHARM++ based and an MPI-based implementation of Jacobi3D. *HPCCG* is distributed as part of the MPI-based Mantevo benchmark suite [15] by Sandia National Laboratories. It mimics the performance of unstructured implicit finite element methods and can scale to large number of nodes. *LULESH* is the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics mini-app [21]. It is a mesh-based physics code on an unstructured hexahedral mesh with element centering and nodal centering. *LeanMD* [17], written in CHARM++, simulates the behavior of atoms based on short-range non-bonded force calculation in NAMD [27]. *miniMD* is part of the Mantevo benchmark suite [15] written in MPI. It mimics the operations performed in LAMMPS. In contrast to the first four benchmarks, the molecular dynamic mini-apps have low memory footprint. Moreover, owing to their implementations, checkpoint data in these programs may be scattered in the memory resulting in extra overheads during operations that require traversal of application data.

Benchmark	Configuration (per core)	Memory Pressure
Jacobi3D	64*64*128 grid points	high
HPCCG	40*40*40 grid points	high
LULESH	32*32*64 mesh elements	high
LeanMD	4000 atoms	low
miniMD	1000 atoms	low

Table 2: Mini-application configuration.

In our experiments, the MPI based programs were executed using AMPI [16], which is Charm++'s interface for MPI programs. The experiments were performed on Intrepid at ANL. Intrepid is an IBM Blue Gene/P with a 3D-torus based high speed interconnect. The configuration of our experiments can be seen in Table 2. The Charm++ and MPI implementation of Jacobi3D used the same configuration in our experiments.

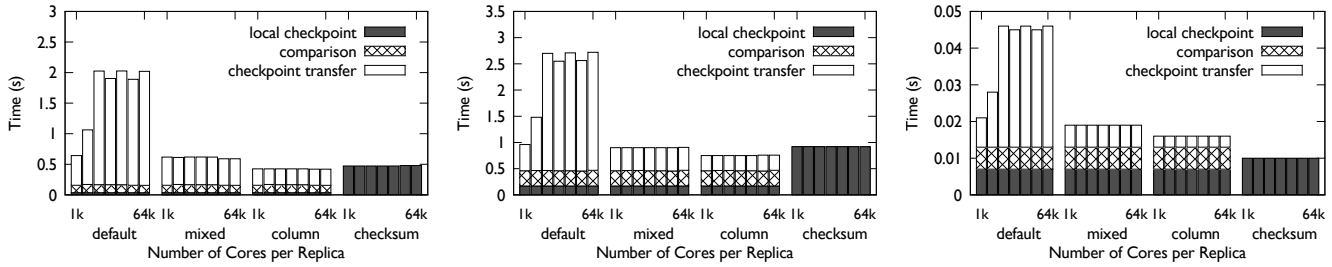
To produce an SDC, our fault injector injects a fault by flipping a randomly selected bit in the user data that will be checkpointed. On most existing systems, when a hard error such as a processor failure occurs, the job scheduler kills the entire job. To avoid a total shutdown, we implement a *no-response* scheme to mimic a 'fail-stop' error. When a hard fault is injected to a node, the process on that node stops responding to any communication. Thereafter, when the buddy node of the this node does not receive heartbeat for a certain period of time, the node is diagnosed as dead.

6.2 Forward Path

In this section, we analyze the overheads ACR incurred in a failure-free case. These overheads include the time spent in local checkpointing, transferring the checkpoints, and comparing the checkpoints. For these experiments, the system size is varied from 2K cores to 128K cores, i.e. there are 1K to 64K cores assigned for each replica.

Figures 8 presents a comparison of the overheads using the default method and with the proposed optimizations for all the mini-apps described. To easily view the change in overheads we graph four of the mini-apps which have higher memory usage on the left and two of the molecular dynamic simulation on the right.

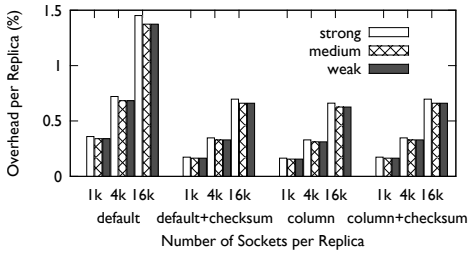
Using the default mapping method, we observe a four-fold increase in the overheads (e.g., from 0.6s to 2s in the case of Jacobi3D) as the system size is increased from 1K cores to 64K cores per replica. By analyzing the time decomposition, we find that the time for inter-replica transfer of the checkpoints keeps increasing while the time spent on local checkpointing and comparison of checkpoints remains constant. An interesting observation is the linear increase of the overheads from 1K to 4K cores and its constancy beyond 4K cores. This unusual increase and steadiness is a result of the change in the length of the Z dimension in the allocated system which determines the load on the bisection links between replica 1 and replica 2 (Section 4.2). As the system size is increased from 1K to 4K cores per



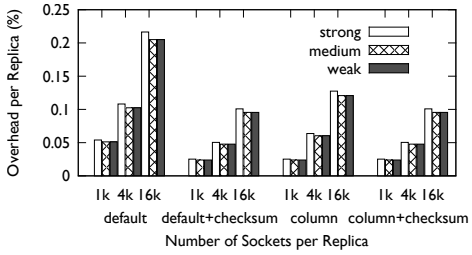
(a) Jacobi3D Charm++ (b) LULESH (c) LeanMD

(d) Jacobi3D AMPI (e) HPCCG (f) miniMD

Figure 8: Single checkpointing overhead. Our framework incurs minimal overheads and provides scalable error detection.


strong
medium
weak
Overhead per Replica (%)
1k 4k 16k 1k 4k 16k 1k 4k 16k 1k 4k 16k
default default+checksum column column+checksum
Number of Sockets per Replica

(a) Jacobi3D Charm++


strong
medium
weak
Overhead per Replica (%)
1k 4k 16k 1k 4k 16k 1k 4k 16k 1k 4k 16k
default default+checksum column column+checksum
Number of Sockets per Replica

(b) LeanMD

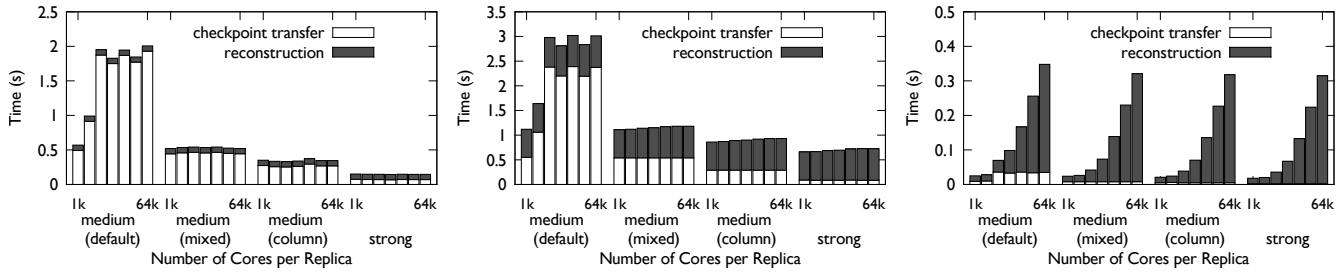
Figure 9: ACR Forward Path Overhead.

replica, the Z dimension increases from 8 to 32, after which it becomes stagnant. Beyond 4K cores, only the X and Y dimensions change but they do not have any impact on the performance. We make use of the mapping schemes proposed in Section 4.2 to eliminate the dependence of overheads of the default method on the length of Z dimension. Figure 8 shows that column and mixed mappings help reduce the inter-replica communication time significantly, enabling the full checkpoint-based error detection method to incur a constant overhead. Moreover, no significant performance difference was found for applications using column or mixed mappings when compared to the default mapping.

In contrast, the overheads incurred using checksum based error detection method remain constant irrespective of the mapping used. Most of the time is spent in computing the checksum with trivial amount of time being spent in checksum transfer and comparison as expected since the checksum data size is only 32 bytes. Note that, due to extra computation cost one has to pay for computing checksum, overheads for it are even larger than the column-mapping for high memory pressure applications. Compared to the other three memory consuming mini-apps, LULESH takes longer time in local checkpointing since it contains more complicated data structures for serialization.

Figure 8c and 8f present the checkpointing overheads for the molecular dynamic mini-apps. While the general trend of results for these mini-apps is similar to the high-memory-pressure mini-apps, the effect of small size of checkpoints and scattered data in memory results in some differences. First, gains in eliminating the overhead due to use of optimal mappings are lower in comparison to the high memory pressure mini-apps. Secondly, only 20% of the time is spent in remote checkpoint transfer with optimal mapping while for the first four mini-apps checkpoint transfer costs around 50% of the time. Thirdly, the checksum method outperforms other schemes though the absolute time is now in $100 - 200$ms range.

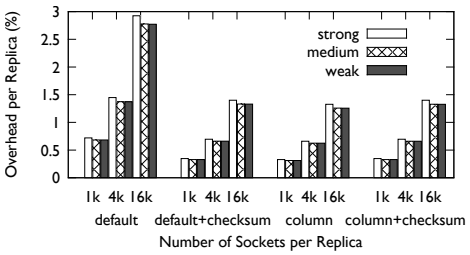
Figure 9 shows the checkpoint overhead of ACR for Jacobi3D and LeanMD when checkpointing at the optimal checkpoint interval according to the model in Section 5. The MTBF for hard error used in the model is 50 years per socket while the SDC rate per socket is estimated as $10,000$ FIT. The low checkpoint overhead enables us to checkpoint more often to reduce the rework overhead. The optimal checkpoint interval for Jacobi3d and LeanMD is 133s and 24s on 16K cores with default mapping. Use of either checksum or topology mapping optimization can bring further down the low checkpointing overhead (1.5%) of default mapping by 50%. Overhead of using strong resilience scheme is



(a) Jacobi3D Charm++ (b) LULESH (c) LeanMD

(d) Jacobi3D AMPI (e) HPCCG (f) miniMD

Figure 10: Single restart overhead. Strong resilience scheme benefits because of the smaller amount of checkpoint data transmitted.



(a) Jacobi3D Charm++

(b) LeanMD

Figure 11: ACR Overall Overhead.

slightly higher than overhead of weak and medium resilience schemes; this is because applications using strong resilience scheme need to checkpoint more frequently to balance the extra rework overhead on hard failures. As the failure rate increases with the number of sockets in the system, forward path overhead also increases.

6.3 Restart from Errors

Figure 10 presents the restart overhead of strong and medium resilience schemes with different mappings. The restart overhead for hard errors includes the time spent on getting the checkpoints from the replica and the time to reconstruct the state from the checkpoint. After an SDC is detected, every node rolls back using local checkpoint without checkpoint transfer, so the restart overhead for SDC is equivalent to the reconstruction part of restarting from hard errors. The only difference between medium and weak resilience is whether an immediate checkpoint is needed (we found the overhead of scheduling an immediate checkpointing to be negligible). Thus the restart overhead is the same for both cases, hence the restart overhead for only medium resilience scheme is presented.

Figure 10 shows that the strong resilience scheme incurs the least restart overhead for all the mini-apps. Two factors help strong resilience scheme outperform the other two schemes- i) the checkpoint that needs to be sent to the crashed replica already exists, and ii) only the buddy of the crashed node has to send the checkpoint to the spare node. Since there is only one inter replica message needed to transfer checkpoints in the strong resilience scheme, we found that mapping does not affect its performance. In comparison, for the medium and the weak resilience schemes, every node in the healthy replica has to send the checkpoint to its buddy in the crashed replica. The simultaneous communication from all the nodes results in network congestion similar to what we saw during checkpointing phase (§ 6.2): the time increase comes from the checkpoint transfer stage as shown in Figure 10. We make use of topology aware mapping to address the congestion problem and bring down the recovery overhead from 2s to 0.41s in the case of Jacobi3D for the medium resilience schemes. Similar results were found for the other benchmarks with relatively large checkpoints.

For LeanMD, which has a small checkpoint, the overheads are presented in Figure 10c. Note that unlike checkpointing, the restart of a crashed replica is an unexpected event. Hence it requires several barriers and broadcasts that are

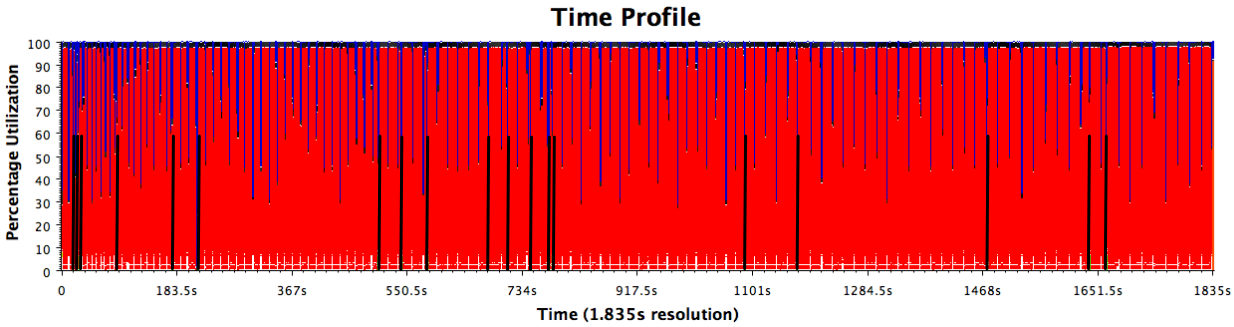


Figure 12: Adaptivity of ACR to changing failure rate. Black lines show when failures are injected. White lines indicate when checkpoints are performed. ACR schedules more checkpoints when there are more failures at the beginning and fewer checkpoints towards the end.

key contributors to the restart time when dealing with applications such as LeanMD whose typical restart time is in tens of milliseconds. Figure 10c shows these effects with a small increase in reconstruction time as the core count is increased. Further inspection confirms that the extra overheads can be attributed to the synchronization costs.

Figure 11 shows the overall overhead of ACR which includes the restart and checkpointing overhead for Jacobi3D and LeanMD at their optimal checkpoint interval. It follows similar trend as shown in Figure 9; the overall overhead is larger than checkpointing overhead alone because of the time spent in recovering from hard failure and SDC. Although restarting is faster using strong resilience as shown in Figure 10, the extra checkpointing overhead and extra time spent re-executing the work lost due to hard failures makes it worse compared to weak and medium resilience no matter which optimization techniques are used. Regardless, *the overhead of strong resilience is less than* 3% *for Jacobi3D and around* 0.45% *for LeanMD. Using optimizations, the overall overhead is further reduced to* 1.4% *and* 0.2%. As such, ACR performs well in comparison to other libraries such as SCR with overhead of 5% [24].

6.4 Adaptivity

As discussed in section 2.2, ACR can dynamically schedule checkpoint based on the failure behaviour. In order to test ACR's capability to adapting to the change of failure rate, we performed a 30 minutes run of Jacobi3D benchmark on 512 cores of BGP with 19 failures injected during the run. The failures are injected according to Weibull process with a decreasing failure rate (shape parameter is 0.6). Figure 12 shows the timeline profile for this run. The red part is the useful work done by application. Black lines mean a failure is injected at that time and white lines indicate that a checkpoint is performed. As can be seen in the figure, more failures are injected at the beginning and the failure rate keeps decreasing as time progresses. ACR changes the checkpoint interval based on the current observed mean time between failures. Accordingly, it schedules more checkpoints in the beginning (checkpoint interval is 6s) and fewer at the end (checkpoint interval increases to 17s).

7. CONCLUSIONS

This paper introduced ACR, an automatic checkpoint/restart framework to make parallel computing systems robust against both silent data corruptions and hard errors. ACR uses 50% of a machine's resources for redundant computation. Such investment is justified in making ACR a general purpose solution for silent data corruptions and in having a resilient solution that outperforms traditional checkpoint/restart in high failure-rate scenarios. ACR aims to automatically recover applications from failures and automatically adjust the checkpoint interval based on the environment. ACR supports three recovery schemes with different levels of resilience. We built a performance model to understand the interaction of SDC and hard errors and explore the trade-off between performance and reliability in the three schemes.

We described the design and implementation of ACR in an established runtime system for parallel computing. We showed the utility of topology aware mapping implemented in ACR, and its impact on the scalability. ACR was tested on a leading supercomputing installation by injecting failures during application execution according to different distributions. We used five mini-apps written in two different programming models and demonstrated that ACR can be used effectively. Our results suggest that ACR can scale to 131,072 cores with low overhead.

Acknowledgment

This research was supported in part by the Blue Waters project (which is supported by the NSF grant OCI 07-25070) and by the US Department of Energy under grant DOE DE-SC0001845. This work used machine resources from PARTS project and Director's discretionary allocation on Intrepid at ANL for which authors thank the ALCF and ANL staff. The authors appreciate the help from Jonathan Lifflander for editing parts of the paper.

8. REFERENCES

[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005.

[2] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Supercomputing*, pages 1 –12, Nov. 2011.

[3] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *JPDC*, 69(4):410–416, 2009.

[4] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Supercomputing*, SC '11, pages 33:1–33:11, New York, NY, USA, 2011. ACM.

[5] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009.

[6] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *Supercomputing*, SC '12, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Comp. Syst.*, 22(3):303–312, 2006.

[8] C. Engelmann, H. H. Ong, and S. L. Scott. The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194. ACTA Press, Calgary, AB, Canada, Feb. 2009.

[9] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Architectural support for programming languages and operating systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[10] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[11] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Supercomputing*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] Fletcher checksum algorithm wiki page. .

[13] S. Genaud, C. Rattanapoka, and U. L. Strasbourg. A peer-to-peer framework for robust execution of message passing parallel programs. In *In EuroPVM/MPI 2005, volume 3666 of LNCS*, pages 276–284. Springer-Verlag, 2005.

[14] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *SciDAC*, 2006.

[15] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratories, September 2009.

[16] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[17] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.

[18] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[19] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan. A study of dynamic meta-learning for failure prediction in large-scale systems. *J. Parallel Distrib. Comput.*, 70(6):630–643, June 2010.

[20] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *Computers, IEEE Transactions on*, 50(7):699–708, 2001.

[21] Lulesh. http://computation.llnl.gov/casc/ShockHydro/.

[22] E. Meneses, X. Ni, and L. V. Kale. A Message-Logging Protocol for Multicore Systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[23] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329 – 335, sept. 2005.

[24] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, pages 1–11, 2010.

[25] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247. IEEE, 2005.

[26] X. Ni, E. Meneses, and L. V. Kalé. Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm. In *IEEE Cluster 12*, Beijing, China, September 2012.

[27] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[28] B. Schroeder and G. Gibson. A large scale study of failures in high-performance-computing systems. In *International Symposium on Dependable Systems and Networks (DSN)*, 2006.

[29] J. Vetter. Hpc landscape application accelerators: Deus ex machina? Invited Talk at High Performance Embedded Computing Workshop, Sep. 2009.

[30] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE Cluster*, pages 93–103, San Diego, CA, September 2004.